Traffic Management

🕐 22 minute read

Istio's traffic routing rules let you easily control the flow of traffic and API calls between services. Istio simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it easy to set up important tasks like A/B testing, canary rollouts, and staged rollouts with percentage-based traffic splits. It also provides out-of-box failure recovery features that help make your

application more robust against failures of dependent services or the network.

Istio's traffic management model relies on the Envoy proxies that are deployed along with your services. All traffic that your mesh services send and receive (data plane traffic) is proxied through Envoy, making it easy to direct and control traffic around your mesh without making any changes to your services.

If you're interested in the details of how the features described in this guide work, you can find out more about Istio's traffic management implementation in the architecture overview. The rest of this guide introduces Istio's traffic management features.

Introducing Istio traffic management

In order to direct traffic within your mesh, Istio needs to know where all your endpoints are, and which services they belong to. To populate its own service registry, Istio connects to a service discovery system. For example, if you've installed Istio on a Kubernetes cluster, then Istio automatically detects the services and endpoints in that cluster.

Using this service registry, the Envoy proxies can then direct traffic to the relevant services. Most microservice-based applications have multiple instances of each service workload to handle service traffic, sometimes referred to as a load balancing pool. By default, the Envoy

proxies distribute traffic across each service's load balancing pool using a round-robin model, where requests are sent to each pool member in turn, returning to the top of the pool once each service instance has received a request.

While Istio's basic service discovery and load balancing gives you a working service mesh, it's far from all that Istio can do. In many cases you might want more fine-grained control over what happens to your mesh traffic. You might want to direct a particular percentage of traffic to a new version of a service as part of A/B testing, or apply a different load balancing policy to traffic for a particular subset of service instances. You might also want to apply special rules to traffic coming into or out of your mesh, or add an external dependency of your mesh to the service registry. You

can do all this and more by adding your own traffic configuration to Istio using Istio's traffic management API.

Like other Istio configuration, the API is specified using Kubernetes custom resource definitions (CRDs), which you can configure using YAML, as you'll see in the examples.

The rest of this guide examines each of the traffic management API resources and what you can do with them. These resources are:

- Virtual services
- Destination rules
- Gateways
- Service entries
- Sidecars

This guide also gives an overview of some of the network resilience and testing features that are built in to the API resources.

Virtual services

Virtual services, along with destination rules, are the key building blocks of Istio's traffic routing functionality. A virtual service lets you configure how requests are routed to a service within an Istio service mesh, building on the basic connectivity and discovery provided by Istio and your platform. Each virtual service consists of a set of routing rules that are evaluated in order, letting Istio match each given request to the virtual service to a specific real destination within the mesh. Your

mesh can require multiple virtual services
or none depending on your use case.

Why use virtual services?

Virtual services play a key role in making
Istio's traffic management flexible and
powerful. They do this by strongly
decoupling where clients send their
requests from the destination workloads
that actually implement them. Virtual
services also provide a rich way of
specifying different traffic routing rules for
sending traffic to those workloads.

Why is this so useful? Without virtual
services, Envoy distributes traffic using
round-robin load balancing between all

service instances, as described in the introduction. You can improve this behavior with what you know about the workloads. For example, some might represent a different version. This can be useful in A/B testing, where you might want to configure traffic routes based on percentages across different service versions, or to direct traffic from your internal users to a particular set of instances.

With a virtual service, you can specify traffic behavior for one or more hostnames. You use routing rules in the virtual service that tell Envoy how to send the virtual service's traffic to appropriate destinations. Route destinations can be versions of the same service or entirely different services.

A typical use case is to send traffic to different versions of a service, specified as

service subsets. Clients send requests to the virtual service host as if it was a single entity, and Envoy then routes the traffic to the different versions depending on the virtual service rules: for example, "20% of calls go to the new version" or "calls from these users go to version 2". This allows you to, for instance, create a canary rollout where you gradually increase the percentage of traffic that's sent to a new service version. The traffic routing is completely separate from the instance deployment, meaning that the number of instances implementing the new service version can scale up and down based on traffic load without referring to traffic routing at all. By contrast, container orchestration platforms like Kubernetes only support traffic distribution based on instance scaling, which quickly becomes complex. You can read more about how

virtual services help with canary deployments in Canary Deployments using Istio.

Virtual services also let you:

- Address multiple application services through a single virtual service. If your mesh uses Kubernetes, for example, you can configure a virtual service to handle all services in a specific namespace. Mapping a single virtual service to multiple "real" services is particularly useful in facilitating turning a monolithic application into a composite service built out of distinct microservices without requiring the consumers of the service to adapt to the transition. Your routing rules can specify "calls to these URIs of monolith.com go to microservice A", and so on. You can see how this works in one of

- Configure traffic rules in combination with gateways to control ingress and egress traffic.

In some cases you also need to configure destination rules to use these features, as these are where you specify your service subsets. Specifying service subsets and other destination-specific policies in a separate object lets you reuse these cleanly between virtual services. You can find out more about destination rules in the next section.

Virtual service example

The following virtual service routes

requests to different versions of a service depending on whether the request comes from a particular user.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: reviews
        subset: v3
```

The hosts field

The `hosts` field lists the virtual service's hosts - in other words, the user-addressable destination or destinations that these routing rules apply to. This is the address or addresses the client uses when sending requests to the service.

```
hosts:
- reviews
```

The virtual service hostname can be an IP address, a DNS name, or, depending on the platform, a short name (such as a Kubernetes service short name) that resolves, implicitly or explicitly, to a fully qualified domain name (FQDN). You can also use wildcard ("*") prefixes, letting you create a single set of routing rules for all matching services. Virtual service hosts don't actually have to be part of the Istio service registry, they are simply virtual

destinations. This lets you model traffic for virtual hosts that don't have routable entries inside the mesh.

Routing rules

The `http` section contains the virtual service's routing rules, describing match conditions and actions for routing HTTP/1.1, HTTP2, and gRPC traffic sent to the destination(s) specified in the hosts field (you can also use `tcp` and `tls` sections to configure routing rules for TCP and unterminated TLS traffic). A routing rule consists of the destination where you want the traffic to go and zero or more match conditions, depending on your use case.

Match condition

The first routing rule in the example has a

condition and so begins with the `match` field. In this case you want this routing to apply to all requests from the user "jason", so you use the `headers`, `end-user`, and `exact` fields to select the appropriate requests.

```
- match:
  - headers:
      end-user:
        exact: jason
```

Destination

The route section's `destination` field specifies the actual destination for traffic that matches this condition. Unlike the virtual service's host(s), the destination's host must be a real destination that exists in Istio's service registry or Envoy won't know where to send traffic to it. This can be a mesh service with proxies or a non-mesh service added using a service entry.

In this case we're running on Kubernetes and the host name is a Kubernetes service name:

```
route:
- destination:
    host: reviews
    subset: v2
```

Note in this and the other examples on this page, we use a Kubernetes short name for the destination hosts for simplicity. When this rule is evaluated, Istio adds a domain suffix based on the namespace of the virtual service that contains the routing rule to get the fully qualified name for the host. Using short names in our examples also means that you can copy and try them in any namespace you like.

Using short names like this only

works if the destination hosts and the virtual service are actually in the same Kubernetes namespace. Because using the Kubernetes short name can result in misconfigurations, we recommend that you specify fully qualified host names in production environments.

The destination section also specifies which subset of this Kubernetes service you want requests that match this rule's conditions to go to, in this case the subset named v2. You'll see how you define a service subset in the section on destination rules below.

Routing rule precedence

Routing rules are **evaluated in sequential order from top to bottom**, with the first rule in the virtual service definition being given highest priority. In this case you want anything that doesn't match the first routing rule to go to a default destination, specified in the second rule. Because of this, the second rule has no match conditions and just directs traffic to the v3 subset.

```
- route:
  - destination:
      host: reviews
      subset: v3
```

We recommend providing a default "no condition" or weight-based rule (described below) like this as the last rule in each virtual service to ensure that traffic to the virtual service always has at least one matching route.

More about routing rules

As you saw above, routing rules are a powerful tool for routing particular subsets of traffic to particular destinations. You can set match conditions on traffic ports, header fields, URIs, and more. For example, this virtual service lets users send traffic to two separate services, ratings and reviews, as if they were part of a bigger virtual service at `http://bookinfo.com/.` The virtual service rules match traffic based on request URIs and direct requests to the appropriate service.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
    - bookinfo.com
  http:
  - match:
    - uri:
        prefix: /reviews
    route:
    - destination:
        host: reviews
  - match:
    - uri:
        prefix: /ratings
    route:
    - destination:
        host: ratings
```

For some match conditions, you can also choose to select them using the exact value, a prefix, or a regex.

You can add multiple match conditions to the same match block to AND your

conditions, or add multiple match blocks to the same rule to OR your conditions. You can also have multiple routing rules for any given virtual service. This lets you make your routing conditions as complex or simple as you like within a single virtual service. A full list of match condition fields and their possible values can be found in the `HTTPMatchRequest` reference.

In addition to using match conditions, you can distribute traffic by percentage "weight". This is useful for A/B testing and canary rollouts:

```
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 75
    - destination:
        host: reviews
        subset: v2
      weight: 25
```

You can also use routing rules to perform some actions on the traffic, for example:

- Append or remove headers.
- Rewrite the URL.
- Set a `retry policy` for calls to this destination.

To learn more about the actions available, see the `HTTPRoute reference`.

Destination rules

Along with virtual services, destination rules are a key part of Istio's traffic routing functionality. You can think of virtual services as how you route your traffic **to** a given destination, and then you use destination rules to configure what happens to traffic **for** that destination. Destination rules are applied after virtual service routing rules are evaluated, so they apply to the traffic's "real" destination.

In particular, you use destination rules to specify named service subsets, such as grouping all a given service's instances by version. You can then use these service subsets in the routing rules of virtual services to control the traffic to different instances of your services.

Destination rules also let you customize Envoy's traffic policies when calling the entire destination service or a particular service subset, such as your preferred load balancing model, TLS security mode, or circuit breaker settings. You can see a complete list of destination rule options in the Destination Rule reference.

Load balancing options

By default, Istio uses a round-robin load balancing policy, where each service instance in the instance pool gets a request in turn. Istio also supports the following models, which you can specify in destination rules for requests to a particular service or service subset.

- Random: Requests are forwarded at random to instances in the pool.
- Weighted: Requests are forwarded to instances in the pool according to a specific percentage.
- Least requests: Requests are forwarded to instances with the least number of requests.

See the `Envoy load balancing documentation` for more information about each option.

Destination rule example

The following example destination rule configures three different subsets for the `my-svc` destination service, with different load balancing policies:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
  - name: v3
    labels:
      version: v3
```

Each subset is defined based on one or more `labels`, which in Kubernetes are key/value pairs that are attached to objects such as Pods. These labels are applied in the Kubernetes service's deployment as

`metadata` to identify different versions.

As well as defining subsets, this destination rule has both a default traffic policy for all subsets in this destination and a subset-specific policy that overrides it for just that subset. The default policy, defined above the `subsets` field, sets a simple random load balancer for the `v1` and `v3` subsets. In the `v2` policy, a round-robin load balancer is specified in the corresponding subset's field.

Gateways

You use a `gateway` to manage inbound and outbound traffic for your mesh, letting you specify which traffic you want to enter or

leave the mesh. Gateway configurations are applied to standalone Envoy proxies that are running at the edge of the mesh, rather than sidecar Envoy proxies running alongside your service workloads.

Unlike other mechanisms for controlling traffic entering your systems, such as the Kubernetes Ingress APIs, Istio gateways let you use the full power and flexibility of Istio's traffic routing. You can do this because Istio's Gateway resource just lets you configure layer 4-6 load balancing properties such as ports to expose, TLS settings, and so on. Then instead of adding application-layer traffic routing (L7) to the same API resource, you bind a regular Istio virtual service to the gateway. This lets you basically manage gateway traffic like any other data plane traffic in an Istio mesh.

Gateways are primarily used to manage ingress traffic, but you can also configure egress gateways. An egress gateway lets you configure a dedicated exit node for the traffic leaving the mesh, letting you limit which services can or should access external networks, or to enable secure control of egress traffic to add security to your mesh, for example. You can also use a gateway to configure a purely internal proxy.

Istio provides some preconfigured gateway proxy deployments (istio-ingressgateway and istio-egressgateway) that you can use - both are deployed if you use our demo installation, while just the ingress gateway is deployed with our default profile. You can apply your own gateway configurations to these deployments or deploy and configure your own gateway proxies.

Gateway example

The following example shows a possible gateway configuration for external HTTPS ingress traffic:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: ext-host-gwy
spec:
  selector:
    app: my-gateway-controller
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - ext-host.example.com
    tls:
      mode: SIMPLE
      credentialName: ext-host-cert
```

This gateway configuration lets HTTPS traffic from `ext-host.example.com` into the

mesh on port 443, but doesn't specify any routing for the traffic.

To specify routing and for the gateway to work as intended, you must also bind the gateway to a virtual service. You do this using the virtual service's gateways field, as shown in the following example:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-svc
spec:
  hosts:
  - ext-host.example.com
  gateways:
  - ext-host-gwy
```

You can then configure the virtual service with routing rules for the external traffic.

Service entries

You use a service entry to add an entry to the service registry that Istio maintains internally. After you add the service entry, the Envoy proxies can send traffic to the service as if it was a service in your mesh. Configuring service entries allows you to manage traffic for services running outside of the mesh, including the following tasks:

- Redirect and forward traffic for external destinations, such as APIs consumed from the web, or traffic to services in legacy infrastructure.
- Define retry, timeout, and fault injection policies for external destinations.
- Run a mesh service in a Virtual Machine (VM) by adding VMs to your mesh.

You don't need to add a service entry for every external service that you want your mesh services to use. By default, Istio configures the Envoy proxies to passthrough requests to unknown services. However, you can't use Istio features to control the traffic to destinations that aren't registered in the mesh.

Service entry example

The following example mesh-external service entry adds the `ext-svc.example.com` external dependency to Istio's service registry:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: svc-entry
spec:
  hosts:
  - ext-svc.example.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
```

You specify the external resource using the `hosts` field. You can qualify it fully or use a wildcard prefixed domain name.

You can configure virtual services and destination rules to control traffic to a service entry in a more granular way, in the same way you configure traffic for any other service in the mesh. For example, the following destination rule configures the traffic route to use mutual TLS to secure

the connection to the `ext-svc.example.com` external service that we configured using the service entry:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ext-res-dr
spec:
  host: ext-svc.example.com
  trafficPolicy:
    tls:
      mode: MUTUAL
      clientCertificate: /etc/certs/myclientcert.pe
m
      privateKey: /etc/certs/client_private_key.pem
      caCertificates: /etc/certs/rootcacerts.pem
```

See the `Service Entry reference` for more possible configuration options.

Sidecars

By default, Istio configures every Envoy proxy to accept traffic on all the ports of its associated workload, and to reach every workload in the mesh when forwarding traffic. You can use a `sidecar` configuration to do the following:

- Fine-tune the set of ports and protocols that an Envoy proxy accepts.
- Limit the set of services that the Envoy proxy can reach.

You might want to limit sidecar reachability like this in larger applications, where having every proxy configured to reach every other service in the mesh can potentially affect mesh performance due to high memory usage.

You can specify that you want a sidecar configuration to apply to all workloads in a

particular namespace, or choose specific workloads using a `workloadSelector`. For example, the following sidecar configuration configures all services in the `bookinfo` namespace to only reach services running in the same namespace and the Istio control plane (currently needed to use Istio's policy and telemetry features):

```
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default
  namespace: bookinfo
spec:
  egress:
  - hosts:
    - "./*"
    - "istio-system/*"
```

See the `Sidecar reference` for more details.

Network resilience and testing

As well as helping you direct traffic around your mesh, Istio provides opt-in failure recovery and fault injection features that you can configure dynamically at runtime. Using these features helps your applications operate reliably, ensuring that the service mesh can tolerate failing nodes and preventing localized failures from cascading to other nodes.

Timeouts

A timeout is the amount of time that an Envoy proxy should wait for replies from a given service, ensuring that services don't

hang around waiting for replies indefinitely and that calls succeed or fail within a predictable timeframe. The Envoy timeout for HTTP requests is disabled in Istio by default.

For some applications and services, Istio's default timeout might not be appropriate. For example, a timeout that is too long could result in excessive latency from waiting for replies from failing services, while a timeout that is too short could result in calls failing unnecessarily while waiting for an operation involving multiple services to return. To find and use your optimal timeout settings, Istio lets you easily adjust timeouts dynamically on a per-service basis using virtual services without having to edit your service code. Here's a virtual service that specifies a 10 second timeout for calls to the v1 subset of the

ratings service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    timeout: 10s
```

Retries

A retry setting specifies the maximum number of times an Envoy proxy attempts to connect to a service if the initial call fails. Retries can enhance service availability and application performance by

making sure that calls don't fail permanently because of transient problems such as a temporarily overloaded service or network. The interval between retries (25ms+) is variable and determined automatically by Istio, preventing the called service from being overwhelmed with requests. The default retry behavior for HTTP requests is to retry twice before returning the error.

Like timeouts, Istio's default retry behavior might not suit your application needs in terms of latency (too many retries to a failed service can slow things down) or availability. Also like timeouts, you can adjust your retry settings on a per-service basis in virtual services without having to touch your service code. You can also further refine your retry behavior by adding per-retry timeouts, specifying the

amount of time you want to wait for each
retry attempt to successfully connect to the
service. The following example configures a
maximum of 3 retries to connect to this
service subset after an initial call failure,
each with a 2 second timeout.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    retries:
      attempts: 3
      perTryTimeout: 2s
```

Circuit breakers

Circuit breakers are another useful mechanism Istio provides for creating resilient microservice-based applications. In a circuit breaker, you set limits for calls to individual hosts within a service, such as the number of concurrent connections or how many times calls to this host have failed. Once that limit has been reached the circuit breaker "trips" and stops further connections to that host. Using a circuit breaker pattern enables fast failure rather than clients trying to connect to an overloaded or failing host.

As circuit breaking applies to "real" mesh destinations in a load balancing pool, you configure circuit breaker thresholds in destination rules, with the settings applying to each individual host in the service. The following example limits the number of concurrent connections for the reviews

service workloads of the v1 subset to 100:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
```

You can find out more about creating circuit breakers in Circuit Breaking.

Fault injection

After you've configured your network, including failure recovery policies, you can

use Istio's fault injection mechanisms to test the failure recovery capacity of your application as a whole. Fault injection is a testing method that introduces errors into a system to ensure that it can withstand and recover from error conditions. Using fault injection can be particularly useful to ensure that your failure recovery policies aren't incompatible or too restrictive, potentially resulting in critical services being unavailable.

Unlike other mechanisms for introducing errors such as delaying packets or killing pods at the network layer, Istio' lets you inject faults at the application layer. This lets you inject more relevant failures, such as HTTP error codes, to get more relevant results.

You can inject two types of faults, both

configured using a `virtual service`:

- Delays: Delays are timing failures. They mimic increased network latency or an overloaded upstream service.
- Aborts: Aborts are crash failures. They mimic failures in upstream services. Aborts usually manifest in the form of HTTP error codes or TCP connection failures.

For example, this virtual service introduces a 5 second delay for 1 out of every 1000 requests to the `ratings` service.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        percentage:
          value: 0.1
        fixedDelay: 5s
    route:
    - destination:
        host: ratings
        subset: v1
```

For detailed instructions on how to configure delays and aborts, see Fault Injection.

Working with your applications

Istio failure recovery features are completely transparent to the application. Applications don't know if an Envoy sidecar proxy is handling failures for a called service before returning a response. This means that if you are also setting failure recovery policies in your application code you need to keep in mind that both work independently, and therefore might conflict. For example, suppose you can have two timeouts, one configured in a virtual service and another in the application. The application sets a 2 second timeout for an API call to a service. However, you configured a 3 second timeout with 1 retry in your virtual service. In this case, the application's timeout kicks in first, so your Envoy timeout and retry attempt has no effect.

While Istio failure recovery features

improve the reliability and availability of services in the mesh, applications must handle the failure or errors and take appropriate fallback actions. For example, when all instances in a load balancing pool have failed, Envoy returns an HTTP 503 code. The application must implement any fallback logic needed to handle the HTTP 503 error code.